SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE

ZAVRŠNI RAD

Matija Rossi

Zagreb, 2013.

UNIVERSITY OF ZAGREB
FACULTY OF MECHANICAL ENGINEERING AND
NAVAL ARCHITECTURE

BACHELOR'S THESIS

Mentor:

Prof. dr. sc. Mario Essert

Student:

Matija Rossi

Zagreb, 2013

I declare that I have made this work independently, using the knowledge acquired during my studies and the cited literature.

I would like to thank my mentor, Professor Mario Essert, for his continuous support before and during the writing of this thesis.

I wish also to thank Professor Domagoj Matijević, from the University of Osijek, that he encouraged me to start this work and for all the time, help and support he gave me. A big thank you also to his assistants and all the other great people I met during my stay in Osijek.

This work would not have been possible without the people from the Loccioni Group from Angeli di Rosora (Ancona), Italy, who gave me the opportunity to work there for three months, and to whom I am endlessly grateful. A huge thank you for the immense help and kindness to all the people I met there, that made my stay a beautiful experience: Giacomo Angione, who was my mentor during my stay there, Luca Lattanzi, Cristina Cristalli, Simonetta Piangerelli, and all the others, which I am not listing here individually only because there are so many of them. I want also to thank in particular to Professor David Scaradozzi, from the Technical University in Ancona, who made this internship possible and who supported and helped me whenever I needed.

Finally, I wish to thank my family, my girlfriend and all my friends for the big support and understanding during my studies.

Matija Rossi

SVEUČILIŠTE U ZAGREBU

FAKULTET STROJARSTVA I BRODOGRADNJE

Središnje povjerenstvo za završne i diplomske ispite

Povjerenstvo za završne ispite studija strojarstva za smjerove:
proizvodno inženjerstvo, računalno inženjerstvo, industrijsko inženjerstvo i menadžment, inženjerstvo materijala i mehatronika i robotika

Sveučilište u Zagrebu Fakultet strojarstva i brodogradnje	
Datum	Prilog
Klasa:	
Ur.broj:	

ZAVRŠNI ZADATAK

Student: **MATIJA ROSSI** Mat. br.: 0035178935

Naslov rada na hrvatskom jeziku: **Paralelni algoritam za simultanu lokalizaciju i mapiranje za industrijskog mobilnog robota**

Naslov rada na engleskom jeziku: **A Parallel Algorithm for Simultaneous Localization and Mapping for an Industrial Mobile Robot**

Opis zadatka:

Paralelna implementacija algoritma za simultanu lokalizaciju i mapiranje (SLAM) na grafičkom procesoru računala važna je tema današnjeg bavljenja matematičara, računalnih stručnjaka i inženjera. SLAM algoritam temelji se na čestičnom filtru, te je na toj razini ostvaren paralelizam. Algoritam je razvijen za industrijskog mobilnog robota, te predstavlja prvi korak prema potpuno autonomnom obavljanju zadataka. Kôd koji se izvršava na GPU (grafičkom procesoru) pisan je u programskom jeziku CUDA C, te je kao biblioteka uključen u glavni program koji upravlja mobilnim robotom, koji je ostvaren u grafičkom jeziku LabVIEW.

U ovom radu potrebno je:

1. Predstaviti SLAM problem
2. Opisati industrijskog mobilnog robota Loccioni Mo.Di.Bot
3. Istražiti Monte Carlo lokalizaciju i FastSLAM algoritam
4. Predstaviti Nvidia CUDA arhitekturu te osnove programskog jezika CUDA C
5. U jeziku CUDA C implementirati algoritam SLAM

Zadatak zadan:

16. studenog 2012.

Zadatak zadao:

prof. dr. sc. Mario Essert

Rok predaje rada:
1. rok: 15. veljače 2013.
2. rok: 11. srpnja 2013.
3. rok: 13. rujna 2013.

Predviđeni datumi obrane:
1. rok: 27., 28. veljače i 1. ožujka 2013.
2. rok: 15., 16. i 17. srpnja 2013.
3. rok: 18., 19., i 20. rujna 2013.

Predsjednik Povjerenstva:

Prof. dr. sc. Zoran Kunica

Abstract

This thesis deals with an implementation of a parallelized simultaneous localization and mapping (SLAM) algorithm based on the Monte Carlo method of particle filtering. It executes on the graphics processing unit (GPU) of a computer using Nvidia CUDA. The parallelization significantly reduces the runtime of the algorithm compared to the classical serial implementation. It has been applied on an industrial quality control mobile robot.

Sažetak

U ovome je radu predstavljen paralelizirani algoritam za simultanu lokalizaciju i mapiranje (SLAM) koji se temelji na Monte Carlo metodi zvanoj čestični filter. Izvršava se na grafičkoj procesorskoj jedinici (GPU) računala koristeći Nvidia CUDA tehnologiju. Paralelizacija značajno smanjuje vrijeme izvršavanja algoritma u usporedbi sa klasičnom serijskom implementacijom. Algoritam je primjenjen na industrijskog mobilnog robota korištenog za kontrolu kvalitete.

Keywords/ključne riječi: mobilni robot; mobile robot; SLAM; simultaneous localization and mapping ; čestični filter; particle filter; Monte Carlo localization; CUDA

Contents

List of Tables

List of Figures

1 Introduction

One of today's mobile robotics main areas of interest is achieving complete robot autonomy. A truly autonomous robot should be capable of executing a given task without requesting from a human operator any details about how to do it. Even from the aspect of navigation there are several problems a mobile robot must be able to solve, as it often has to operate in unknown environments, that can not be fully controlled or it is not convenient to do so. Those problems can be roughly categorized as shown in Figure 1.1:

- Environment mapping
- Robot localization
- Path planning

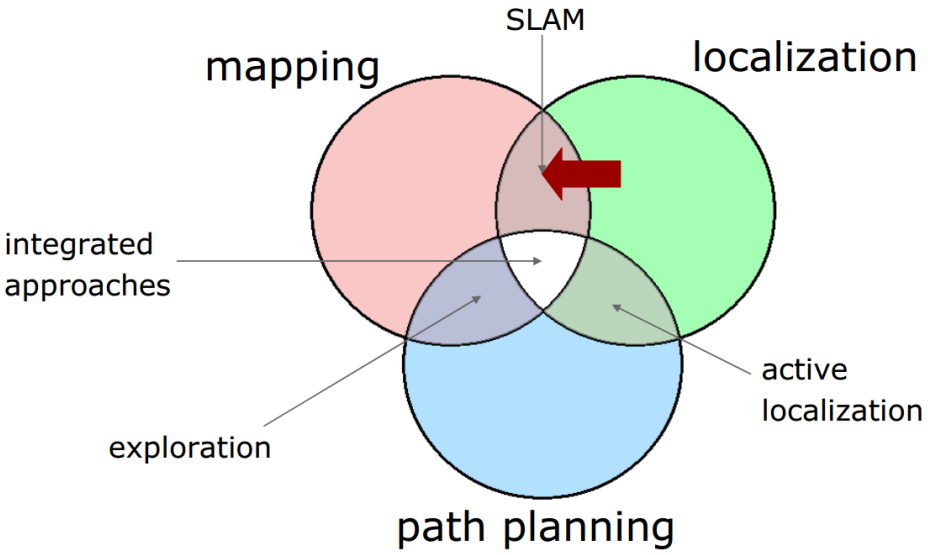


Figure 1.1: Autonomous navigation problems

The localization and the mapping problem together form the so called SLAM problem, an acronym for Simultaneous Localization And Mapping.

This can be described as the problem of building a map of landmarks[1], or improving an existing one, while at the same time localizing the robot within the created map. Solving the SLAM problem was a big achievement in the past decade, and in today's mobile robotics SLAM algorithms are a very important research area.

Having good autonomous mapping removes the need for initial knowledge of the environment, giving the robot the ability to operate every time in a different environment, while an efficient localization technique is needed for the following reasons:

- Robot motion models are approximative
- Odometry (dead reckoning) errors are cumulative
- Inertial sensor errors are cumulative
- GNSS[2] positioning is not accurate and the signal is not always available

Up to date various SLAM techniques have been developed, mainly based on the Extended Kalman Filter (EKF) [2]. However, even if it is still the dominant approach, it has some drawbacks, which are going to be mentioned later in the text. Newer methods are based on Bayesian reasoning, and can achieve better results.

In this paper a simple implementation of a particle filter based SLAM algorithm will be discussed. It has been developed for the Loccioni Mo.Di.Bot, a mobile robot for industrial applications. First it has been implemented using National Instruments LabVIEW, and afterwards it has been parallelized to achieve better performance, combining LabVIEW with Nvidia CUDA.

[1] Features of an environment.

[2] Global Navigation Satellite System (e.g. GPS, GLONASS, Galileo).

1.1 The robot

Figure 1.2: Loccioni Mo.Di.Bot

The Loccioni Mo.Di.Bot is a robot for product quality control in life-cycle testing laboratories. It's task is to move to the chosen product and perform manipulation and measurements on it with it's on-board equipment. At the moment the robot's path planning must be performed manually for each mission and the environment must be well known in advance, but the final goal is achieving full autonomy and flexibility. This means that the user should only select the desired objects to be inspected, while the robot must be able to navigate in an unknown environment, find objects and create a map for further path planning and localization. The algorithm described

here should be a small step towards that goal.

The main robot's parts can be classified in two categories, accordingly to their purpose (Table 1.1).

Table 1.1: Robot's components

Mobile platform	Measuring & testing equipment
Robusoft RobuLAB80 mobile robot SICK LIDAR 2 Microsoft Kinects	Robotic arm with 7 DOF 3 fingers with pressure sensors Stereo camera Microphone Laser vibrometer

For the control of the robot, there are two PCs, one for each functional category (Table 1.2).

Table 1.2: Control units

Windows PC	Ubuntu PC
LabVIEW ↓ Robot's navigation	ROS (Robot Operating System) ↓ Arm and fist operation

The interesting part for this work is the one regarding the navigation of the mobile platform. The only sensor used here for localization and mapping is the LIDAR[3] device, with the specifications defined in Table 1.3.

[3]Light Detection And Ranging

Table 1.3: LIDAR specifications

SICK LIDAR	
Range	7 m
Angle of view	180°
Radial resolution	1 cm
Angular resolution	0.5°

The LIDAR has also a very useful feature: it can distinguish between reflective and unreflective surfaces. For each of the 360 measurements in one scan, it associates a boolean value that describes the type of the observed surface: `true` if it is reflective, `false` if it is not. To exploit this features, the landmarks used for localization and mapping are reflective poles positioned across the operational area. This makes landmark detection much easier, but it's not applicable when the environment features can't be manipulated.

2 SLAM

As it was mentioned before, the SLAM problem consists of localization within an unknown environment, while simultaneously building a map of that environment. Because of it's *chicken-or-egg* nature, the SLAM problem is not trivial: if the robot's path (it's position at any time) was known, it would be a straight-forward task to build a map of the environment based on the robot's observations, or if the entire map was known *a priori*, it would not be difficult to locate the robot within it.

During the exploration, the robot executes controls[4] (performs actions) and makes observations of the surrounding environment, and both operations are affected by noise. The errors resulting from this are strongly tied together: while the robot is in motion it accumulates error on the belief of it's position, so the environment observed by the robot's sensors is not only affected by the measurement error, but also by this uncertainty over the position. Because of this correlation, it is impossible to make any good estimation of the map without also estimating the robot's pose.

At any moment, the robot maintains a set of hypothesis about it's position and the position of objects around it. Those beliefs are updated after every action or observation, according to measurements from various robot's sensors, which is a process called filtering. Both the motion model (controls with noise) and the measurement model (observations with noise) represent probabilistic constraints: the motion model is a constraint for the robot's pose in the next step, and the measurement model is a constraint for the robot's pose relative to an observed feature of the environment (landmark). They are used to estimate the robot's pose and the map of the environment,

[4]Translational and rotational velocities set to the robot.

and, as the algorithm evolves, the number of such constraints grows. At first the constraints are not very strong, but as the number of times a landmark is observed increases, the constraints tighten, and if the landmark could be observed an infinite number of times, the estimated position of the robot and of the landmark would exactly match the real one.

It was also said in the introduction that the main approach to solving the SLAM problem is the use of the Extended Kalman Filter (EKF). In this way, the whole map of landmarks and the estimated robot position are estimated by a single EKF. The main drawback of this approach is it's quadratic complexity $\mathcal{O}\left(K^2\right)$, where K is the number of landmarks. This method is also sensitive to wrong data association, meaning that if an observation by the robot is associated to a wrong landmark from the existing map, it can ruin all future estimations of the EKF.

Some of the newer and more efficient algorithms for on-line SLAM are FastSLAM [3] and FastSLAM 2.0 [4], which this work is based on. They both use particle filtering, a method that is already well adopted in robot localization [5, 6]. A basic implementation, like the one in this paper, has a complexity of $\mathcal{O}\left(MK\right)$, where M is the number of particles.

2.1 Bayes and particle filtering

The main purpose of filtering is to continuously estimate a variable of interest as it evolves over time. The Bayes filter (also known as recursive Bayesian estimation) estimates the variable with a posterior probability distribution function, that usually is non-Gaussian and potentially multimodal. The posterior distribution is a probability distribution function that represents our estimation of the variable of interest considering the previous belief (prior distribution) and taking into account some new relevant evidence[5].

One practical implementation of the Bayes filter is by Monte Carlo[6] simulation, with a method called particle filtering. The posterior distribution is represented by a finite number of samples called *particles*. Those particles are copies of the variable of interest, with different states. Each particle is associated to a weight that is relative to the quality of that particle's state.

The particle filter algorithm is recursive and has two main phases:

1. Prediction — After any action that modifies the state of the variable of interest, each particle is updated accordingly to an existing approximative model with the addition of an amount of noise, in order to simulate the real noise on the variable.

2. Update — Each particle's weight is corrected or re-evaluated considering latest sensory information. Each particle simulates measurements accordingly to a measurement model and with an amount of noise. The particle with the most similar simulated measurements to the real sensory data will get the highest weight.

[5] e.g. an observation of the environment.

[6] Algorithms that rely on repeated random sampling to compute results.

If there is a significant diversity within the particle set, a new generation of particles can be selected, in a process called resampling. The probability of each particle to be reselected is it's normalized weight. It also makes sense to just eliminate the particles with weights lower than a threshold. The resampling method can significantly affect the performance of the algorithm.

2.2 Monte Carlo localization

Particle filtering has many applications in mobile robotics, and the most relevant one for the purpose of this text is Monte Carlo localization. It will be explained in this section, along with the principles of particle filtering, as it is the basis of this SLAM implementation. This will serve as an introduction to the SLAM problem definition in subsection 2.3.

Monte Carlo localization solves the problem of continuously estimating the robot's position, but (unlike in SLAM) the environment map is exactly known *a priori*. This estimation problem can be formulated in state-space [7]. For a two-dimensional localization problem, the robot's pose at time t can be represented by the state vector $\mathbf{s}_t = [x_t, y_t, \vartheta_t]^T$. The vector contains the robot's x and y coordinates and it's orientation (heading direction) ϑ. As the robot executes a control u_t in the time interval $\langle t-1, t]$, it's pose is modified. The robot's motion is described by a transition function f_u that describes how u_t modifies the pose and how it is affected by noise v_t:

$$\mathbf{s}_t = \mathrm{f}_u\left(\mathbf{s}_{t-1}, v_t\right)$$

After executing a control, the robot performs an observation of the environment. All the sensor measurements at time t are included in a vector $\mathbf{z}_t$. The measurements are also affected by noise w_t and are related to the state $\mathbf{s}_t$ with the measurement function f_z:

$$\mathbf{z}_t = \mathrm{f}_z\left(\mathbf{s}_t, w_t\right)$$

The objective is to estimate the unknown state $\mathbf{s}_t$ based on a sequence of observations $\mathbf{z}^t$ that have been performed after each action from the sequence $\mathbf{u}^t$. The superscript ^{t} indicates a set of variables from time 1 to t.

The Bayesian approach to this problem is to find the posterior probability distribution that represents the probability that the robot's position is $\mathbf{s}_t$ for all possible positions $\mathbf{s}_t$, given the history of measurements $\mathbf{z}^t$ and controls $\mathbf{u}^t$. It can be written with the conditional probability notation:

$$\mathrm{p}\left(\mathbf{s}_t|\mathbf{z}^t, \mathbf{u}^t\right)$$

Knowing the initial distribution $\mathrm{p}\left(\mathbf{s}_0\right)$, that can be uniform for a completely unknown initial position $\mathbf{s}_0$, the Bayes filter can theoretically calculate this through the phases described in subsection 2.1:

1. Prediction — This step predicts the prior probability distribution[7] of the pose $\mathbf{s}_t$ given the previous pose $\mathbf{s}_{t-1}$ and the control $\mathbf{u}_t$ that changed it. The prior distribution is the sum (integral) of the products of the motion model $\mathrm{p}\left(\mathbf{s}_t|\mathbf{u}_t, \mathbf{s}_{t-1}\right)$ and the posterior distribution $\mathrm{p}\left(\mathbf{s}_{t-1}|\mathbf{z}^{t-1}, \mathbf{u}^{t-1}\right)$ of step $t-1$, over all possible $\mathbf{s}_{t-1}$.

$$\mathrm{p}\left(\mathbf{s}_t|\mathbf{z}^{t-1}, \mathbf{u}^t\right) = \int \mathrm{p}\left(\mathbf{s}_t|\mathbf{u}_t, \mathbf{s}_{t-1}\right) \mathrm{p}\left(\mathbf{s}_{t-1}|\mathbf{z}^{t-1}, \mathbf{u}^{t-1}\right) \mathrm{d}\,\mathbf{s}_{t-1}$$

2. Update — The state of the robot can not be directly measured, instead the environment can be observed. The sensor measurements are projections of the robot's state generated via the probabilistic law $\mathrm{p}\left(\mathbf{z}_t|\mathbf{s}_t\right)$, known as the measurement model. It represents the likelihood of making the observation $\mathbf{z}_t$ given that the robot's position is $\mathbf{s}_t$. The new posterior distribution can then be calculated using Bayes' rule, that is a law explaining how to change existing beliefs in the light of new evidence:

$$\text{posterior} = \eta \cdot \text{likelihood} \cdot \text{prior}$$

[7] The estimated probability distribution before observations are taken into account.

From the above, the new posterior is:

$$\mathrm{p}\left(\mathbf{s}_t|\mathbf{z}^t, \mathbf{u}^t\right) = \eta \cdot \mathrm{p}\left(\mathbf{z}_t|\mathbf{s}_t\right) \cdot \mathrm{p}\left(\mathbf{s}_t|\mathbf{z}^{t-1}, \mathbf{u}^t\right)$$

In this equations η is just a normalizing constant that ensures the posterior distribution adds up to 1.

The Bayesian framework is only conceptual, as closed form solutions for calculating the posterior are only known for specialized cases. For the linear Gaussian estimation problem, that is when the motion model $\mathrm{p}\left(\mathbf{s}_t|\mathbf{u}_t, \mathbf{s}_{t-1}\right)$ and the measurement model $\mathrm{p}\left(\mathbf{z}_t|\mathbf{s}_t\right)$ are linear and the noise probability distribution functions are Gaussian, the posterior distribution $\mathrm{p}\left(\mathbf{s}_t|\mathbf{z}^t, \mathbf{u}^t\right)$ is also Gaussian, and the Kalman filter provides an optimal solution [8, 9]. For non-linear Gaussian systems, the extended Kalman filter (EKF) provides the result by linearizing the motion and measurement models [2, 10]. However, these techniques are limited to cases where the linear Gaussian assumption is a valid approximation. The two main problems are how to represent an arbitrary distribution function with finite computer storage and how to perform the integrations for updating the posterior distribution [5].

As mentioned in subsection 2.1, particle filtering is a method that approximates general probability distribution functions with a finite number of particles, that are differently weighted samples of the distribution. The concept can be illustrated on the most representative case of localization: when there is not any knowledge about the robot's initial position $\mathbf{s}_0$. In that case the particle filter is initiated randomly generating a selected number of particles M, representing copies of the robot, uniformly across the whole map, all with the same weight. After each iteration the distribution of particles will be more and more nonuniform. Areas of the map with higher probability will have a denser concentration of particles, while areas where the robot is

unlikely to be will have a low concentration. This approximated probability distribution function can be multimodal and of arbitrary complexity. If there were an infinite number of particles, the probability distribution function would be continuous across the entire map.

This concept is demonstrated in Figure 2.1, on an example of Monte Carlo localization from the University of Washington. The small red dots are the particles,the bigger green one represent the estimate of the robot's position and the blue beams are sonar readings. The pose estimate in this example is calculated as the weighted mean of all the particles. In the initial state (Figure 2.1a), the position is unknown, so the particles are created randomly, uniformly distributed across the map. For this reason the pose estimate in the initial steps is completely unreliable. However, with the robot constantly moving and measuring, the particles tend to concentrate around areas with high probability. It can be seen how a group of particles from Figure 2.1e to Figure 2.1f gets eliminated. Particles from that group suddenly lost their weight due to new observations when the robot entered the room, so they have not survived the resampling process.

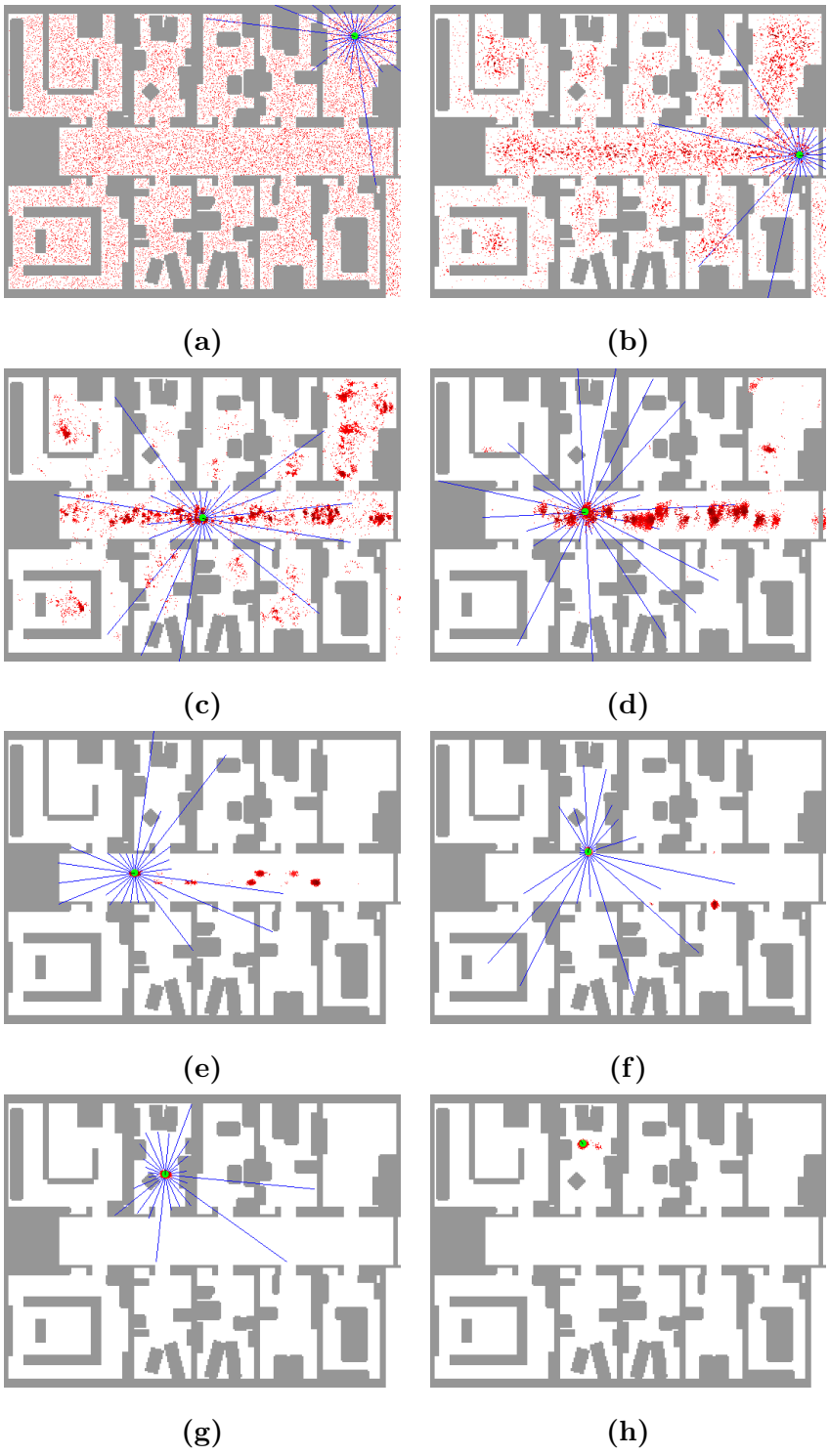


Figure 2.1: MCL example from the University of Washington

2.3 SLAM problem definition

The main difference here is that in SLAM the map of the environment is not known *a priori*. As the robot moves, it builds a map of the explored area. The created map is then used for localization, as described in subsection 2.2, with the addition that the estimation of the map gets also updated after every observation of the environment.

As explained, the poses of the robot evolve according to a motion model, which usually is a time-invariant probabilistic approximation of robot's kinematics:

$$\mathrm{p}\left(\mathbf{s}_t|\mathbf{u}_t, \mathbf{s}_{t-1}\right)$$

This represents the prior distribution of the robot's pose.

To map the environment, the robot searches for landmarks with it's sensors. In a planar problem, landmarks can be considered points on a plane, so each one is characterized just by it's x and y position. Each landmark is denoted $\boldsymbol{\theta}_k$ for $k \in \{1, \ldots, K\}$ where K is the number of landmarks on the map. When the robot observes a landmark, it must either associate it to a landmark that has been previously observed and is present in the map, or it must add a new landmark to the map (list of landmarks). Due to measurement errors and the uncertainty of the actual pose (as it was previously described), the measurement does not precisely correspond to the real situation, so the measurement model is also governed by a probabilistic law:

$$\mathrm{p}\left(\mathbf{z}_t|\mathbf{s}_t, \boldsymbol{\Theta}, n_t\right)$$

The measurement $\mathbf{z}_t$ is a probability function of the actual pose, the set of previously observed landmarks (the map) $\boldsymbol{\Theta} = \{\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_K\}$ and the index

of the detected landmark $n_t \in \{1, \dots, K\}$, that simply describes to which landmark of the map the measurement corresponds. For the SLAM problem definition we assume that the landmarks are identifiable, so this last variable n_t, called correspondence, is known. In real world applications the landmarks usually are not distinguishable one from another, so it is not always easy to decide which landmark a measure belongs to. This is known as the problem of data association, which will be discussed later.

At this point the general definition of the SLAM problem can be formulated. Most general, SLAM is the problem of determining the location of all the landmarks $\mathbf{\Theta}$ and robot's path $\mathbf{s}^t = \{\mathbf{s}_1, \dots, \mathbf{s}_t\}$ from measurements $\mathbf{z}^t = \{\mathbf{z}_1, \dots, \mathbf{z}_t\}$ and controls $\mathbf{u}^t = \{\mathbf{u}_1, \dots, \mathbf{u}_t\}$ [3]. This definition can be expressed by the SLAM posterior distribution:

$$\mathrm{p}\left(\mathbf{s}^t, \mathbf{\Theta} | \mathbf{z}^t, \mathbf{u}^t\right)$$

When the correspondences are known the problem is simpler:

$$\mathrm{p}\left(\mathbf{s}^t, \mathbf{\Theta} | \mathbf{z}^t, \mathbf{u}^t, n^t\right)$$

2.3.1 SLAM problem factorization

The SLAM problem can be decomposed into a robot localization problem and a number of landmark estimation problems that are conditioned on the estimated robot pose[11]. All the landmark estimation problems are independent, as remarked in [12], if the robot's path s^t and correspondence variables n^t are known. This factorization of the SLAM posterior can be written as:

$$\mathrm{p}\left(\mathbf{s}^t, \boldsymbol{\Theta} | \mathbf{z}^t, \mathbf{u}^t, n^t\right) = \underbrace{\mathrm{p}\left(\mathbf{s}^t | \mathbf{z}^t, \mathbf{u}^t, n^t\right)}_{\text{Localization}} \cdot \underbrace{\mathrm{p}\left(\boldsymbol{\Theta} | \mathbf{s}^t, \mathbf{z}^t, \mathbf{u}^t, n^t\right)}_{\text{Landmark estimation}}$$

In this way the first problem is Monte Carlo localization, after which the second problem of landmark estimation (mapping) can be computed efficiently since the robot's pose is known. In total there will be $K + 1$ problems: 1 localization and K problems of estimating the locations of K landmarks conditioned on the path estimate. This can be represented by factorizing the mapping problem to separate problems for each landmark $\boldsymbol{\theta}_k$:

$$\mathrm{p}\left(\mathbf{s}^t, \boldsymbol{\Theta} | \mathbf{z}^t, \mathbf{u}^t, n^t\right) = \underbrace{\mathrm{p}\left(\mathbf{s}^t | \mathbf{z}^t, \mathbf{u}^t, n^t\right)}_{1 \text{ problem}} \cdot \underbrace{\prod_{k=1}^{K} \mathrm{p}\left(\boldsymbol{\theta}_k | \mathbf{s}^t, \mathbf{z}^t, \mathbf{u}^t, n^t\right)}_{K \text{ problems}}$$

3 The algorithm

In this section the SLAM algorithm will be presented, as implemented on the Mo.Di.Bot.

3.1 Initialization

In the initialization step M particles are created. In the described Monte Carlo localization problem, particles could have been generated uniformly across the map, as it was exactly known from the beginning and the only problem was to find where the robot is within it. In the SLAM problem it is not possible to do so, because there is not any knowledge about how the environment looks, so there is no predefined absolute coordinate system. The solution in this case is to set the local coordinate system of the robot as the global system. This is done the moment the algorithm starts, by generating the initial set $\mathbf{P}^0$ of M particles, all in position $\mathbf{s}_0^{[m]}$: $x = 0$, $y = 0$ and $\vartheta = \frac{\pi}{2}$ (oriented towards the positive direction of Y axis).

Algorithm 3.1 Initialization

Require: Number of particles: M

for $i \leftarrow 1 \rightarrow M$ **do** ▷ Create particles

$s^i \leftarrow \left[0, 0, \frac{\pi}{2}\right]^{\mathrm{T}}$

end for

3.2 New pose estimation

The first step of the algorithm is the estimation of the new robot's position after a control modifies it's previous pose. The robot's kinematics are represented with a differential model (Figure 3.1), from which the angle φ' and the distance S' the robot have theoretically passed in the time interval $\langle t-1, t]$ are calculated. The input for this calculation are the angles of rotation of the left and the right wheel (φ_l and φ_r), measured by the incremental encoders on the motors. Knowing the distance L between the wheels and the wheel radius R, S' and φ' can be easily calculated:

$$\begin{aligned} S' &= \frac{R \cdot (\varphi_r + \varphi_l)}{2} \\ \varphi' &= \frac{R \cdot (\varphi_r - \varphi_l)}{L} \end{aligned}$$

It is very important to note that the position of each particle is not updated just by the kinematic model, but also adding Gaussian noise $N_l^{[m]}$ and $N_r^{[m]}$ proportionally to the angles of rotation φ_l and φ_r. In this way the particles get slightly apart one from another, representing the uncertainty introduced by the motion. The passed distance $S^{[m]}$ and angle $\varphi^{[m]}$ of each particle are calculated:

$$\begin{aligned} S^{[m]} &= \frac{R \cdot \left(\varphi_r + \varphi_l + N_r^{[m]}\varphi_r + N_l^{[m]}\varphi_l\right)}{2} \\ \varphi^{[m]} &= \frac{R \cdot \left(\varphi_r - \varphi_l + N_r^{[m]}\varphi_r - N_l^{[m]}\varphi_l\right)}{L} \end{aligned}$$

The new position $\mathbf{s}_t = [x_t, y_t, \vartheta_t]^{\mathrm{T}}$ of each particle can be calculated from S, φ and the previous position $\mathbf{s}_{t-1} = [x_{t-1}, y_{t-1}, \vartheta_{t-1}]^{\mathrm{T}}$:

$$\begin{aligned} x_t^{[m]} &= x_{t-1}^{[m]} + S \cdot \cos\left(\vartheta_{t-1}^{[m]} + \frac{\varphi^{[m]}}{2}\right) \\ y_t^{[m]} &= y_{t-1}^{[m]} + S \cdot \sin\left(\vartheta_{t-1}^{[m]} + \frac{\varphi^{[m]}}{2}\right) \\ \vartheta_t^{[m]} &= \vartheta_{t-1}^{[m]} + \varphi \end{aligned}$$

It is assumed that the wheels maintain constant velocities between two algorithm iterations.

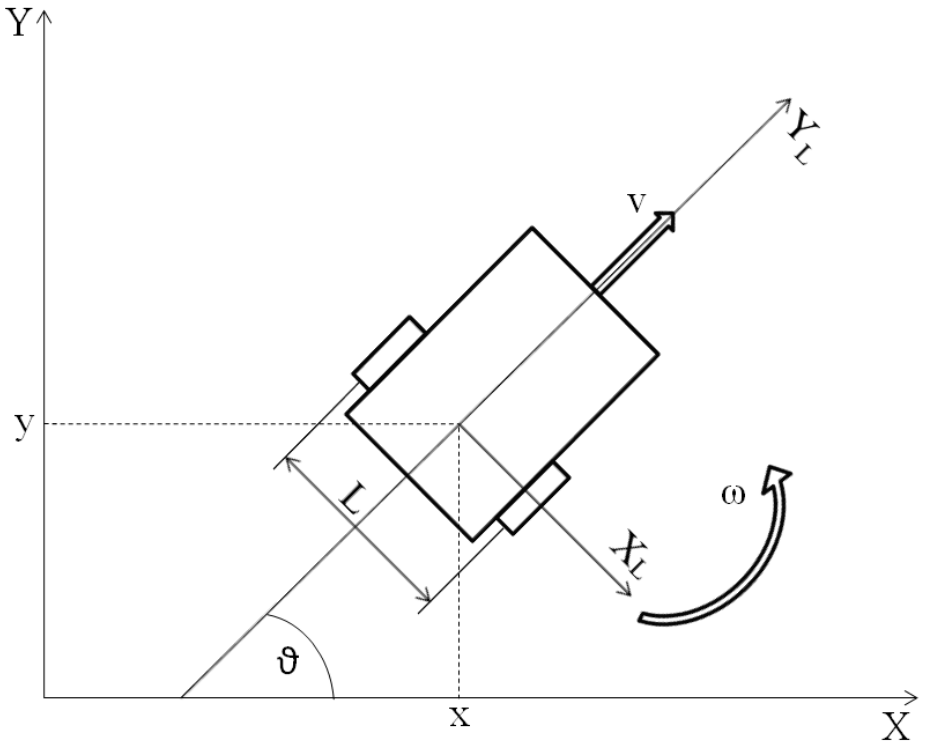


Figure 3.1: Differential kinematic model

Algorithm 3.2 Motion

Require: Set of M particles: P

Left and right wheel angles of rotation: φ_l, φ_r

for $i \leftarrow 1, M$ **do** ▷ For each particle from the set

$N_r \leftarrow \varphi_r \cdot \textsc{RandGauss}(\mu, \sigma)$

$N_l \leftarrow \varphi_l \cdot \textsc{RandGauss}(\mu, \sigma)$

$S \leftarrow R \cdot (\varphi_r + \varphi_l + N_r + N_l)\,/2$

$\varphi \leftarrow R \cdot (\varphi_r - \varphi_l + N_r - N_l)\,/L$

$s^i_{\mathrm{x}} \leftarrow s^i_{\mathrm{x}} + S \cdot \cos\left(s^i_\vartheta + \varphi/2\right)$

$s^i_{\mathrm{y}} \leftarrow s^i_{\mathrm{y}} + S \cdot \sin\left(s^i_\vartheta + \varphi/2\right)$

$s^i_\vartheta \leftarrow s^i_\vartheta + \varphi$

end for

3.3 Landmark locations update

Each particle has it's own list of landmarks, so the particle set $\mathbf{P}$ at time t can be represented as:

$$\mathbf{P}_t = \{\mathbf{s}^{t,[m]}, \boldsymbol{\theta}_1^{[m]}, \ldots, \boldsymbol{\theta}_K^{[m]}\}$$

where $\boldsymbol{\theta}_k^{[m]}$ is the k-th landmark geometrically represented with x and y coordinates, and $m \in \{1, \ldots, M\}$ is the index of the particle.

In this step the algorithm takes new measurements z_t and using them updates the map. A measurement describing a landmark consists of:

1. The angle between the robot's heading direction and the detected landmark
2. The distance between the origin of the robot's local coordinate system and the landmark

After filtering the raw sensory data, when this information becomes available, the algorithm performs data association. It is the operation of assigning the detected landmark either to a previously observed one or creating a new one in the map. This operation is performed independently for every particle. Considering that the particle's pose is known, the Euclidean distance between the observed landmark and an existing one can be calculated. If the distance is within a selected range, the observed landmark is considered to be the existing one, and the pose of the existing landmark is updated by calculating the arithmetic mean of all it's observations, including the latest one. The detected landmark is considered a new one when it is not close enough to any landmark in the particle's map.

The algorithm keeps track of the number of times a landmark has been

observed and the time interval between sightings. From this information it eliminates "false" landmarks created by measurement errors. If a landmark was seen in one or a few consecutive iterations, and then is not observed for a certain number of iterations, it is eliminated, as it probably is nonexistent.

Algorithm 3.3 Map update

Require: Set of M particles: P
Set of N_z measurements: z

```
for i ← 1, M do
    for j ← 1, N_z do                          ▷ For each new measurement
        isAssociated ← false
        for k ← 1, K do                        ▷ For each landmark of the map
            if Distance(z_j, θ^i_k) < value then
                RefreshPosition(θ^i_k, z_j)
                isAssociated ← true
                break
            end if
        end for
        if isAssociated = false then
            θ^i_{K+1} ← z_j                    ▷ Add the new landmark to the map
        end if
    end for
end for
```

3.4 Particle weighting

After the robot performed an action, as described in subsection 3.2, the uncertainty over it's pose increased. To reduce it, latest measurements z_t must be taken into account. The SLAM posterior is represented by weighted particles, so to improve it the weights have to be re-evaluated. The calculation of the particle's weight $w_t^{[m]}$, also called the importance factor, is done with a Gaussian function:

$$w_t^{[m]} = w_{t-1}^{[m]} \cdot \frac{\exp\left(-\frac{1}{2}\frac{\Delta\alpha^{[m]}}{\sigma_\alpha}\right)}{\sigma_\alpha\sqrt{2\pi}} \cdot \frac{\exp\left(-\frac{1}{2}\frac{\Delta r^{[m]}}{\sigma_\mathrm{r}}\right)}{\sigma_r\sqrt{2\pi}}$$

Each particle simulates measurements according to the measurement model. It simply calculates the angle and distance between the particle and landmarks from it's map. When simulating measurements it's important to use the same limitations the robot's sensor has, eg. the view angle, range, etc. In the previous equation $\Delta\alpha^{[m]}$ and $\Delta r^{[m]}$ are the differences between the simulated measurements $z_t^{[m]}$ of particle m and the actual measurements z_t performed by the robot's sensor. The first is the difference in the angle between the robot (particle) and a landmark, the second is the difference in measured distance. Those differences are calculated for every landmark the robot has observed in the current iteration.

$$\begin{aligned}\Delta\alpha^{[m]} &= \arccos\left(\cos(z_{\alpha,t}^{[m]}) \cdot \cos(z_{\alpha,t}) + \sin(z_{\alpha,t}^{[m]}) \cdot \sin(z_{\alpha,t})\right)\\ \Delta r^{[m]} &= z_{\mathrm{r},t}^{[m]} - z_{\mathrm{r},t}\end{aligned}$$

Standard deviations σ_α and σ_r are determined experimentally.

After being calculated, weights have to be normalized to represent probabilities (they must add up to 1). Those probabilities will be used to calculate

the weighted mean of the particle set (this way estimating the robot's position) and they will be also used in the resampling process. The weighted mean of the robot's pose at time t is:

$$\bar{s}_t = \sum_{m=1}^{M} s_t^{[m]} \cdot w_t^{[m]}$$

Algorithm 3.4 Particle weighting

Require: Set of M particles: P
Set of N_z measurements: z

$wSum \leftarrow 0$
$\bar{s} \leftarrow 0$
for $i \leftarrow 1, M$ **do**
  $zSim^i \leftarrow \textsc{MeasSim}(s^i, \Theta^i)$ ▷ Array of simulated measurements
  for $j \leftarrow 1, N_z$ **do**
    $w \leftarrow w \cdot \textsc{Gaussian}(z_j, zSim_j^i)$
  end for
  $wSum \leftarrow wSum + w$
end for
for $i \leftarrow 1, M$ **do**
  $w_{\mathrm{N}}^i \leftarrow w^i / wSum$
  $\bar{s} \leftarrow \bar{s} + s^i \cdot w_{\mathrm{N}}^i$ ▷ Pose estimate
end for

3.5 Resampling

Resampling is a very important part of the algorithm because in case of a bad particles distribution it helps to achieve convergence. Resampling is also potentially dangerous since important samples can get lost. The key question therefore is when to resample. The most general answer is that resampling should be performed when particles have significantly different weights. This can be measured by the variance of the particle weights N_{eff}:

$$N_{\text{eff},t} = \frac{1}{\sum_{m=1}^{M} \left(w_t^{[m]} \right)^2}$$

For equal weights, N_{eff} is maximal and equal to the number of particles M. In this implementation, resampling is performed when $N_{\text{eff}} < \frac{1}{2}M$.

Selecting which particle will "survive" the resampling process and enter into the next generation is done by a discrete random variable. The probability each particle has to be chosen is it's normalized weight. The weights are used to build a binary tree. Using a binary tree improves the performance of the algorithm: every step of a binary search halves the number of items to check in each iteration, so the computational complexity is $\mathcal{O}(\log_2 M)$, opposed to the classical way of linear searching which has a complexity of $\mathcal{O}(M)$. Figure 3.2 shows an example of a tree when the set has $M = 8$ particles, in which case $\log_2 8 = 3$ steps are necessary to find out in which leaf[8] falls a random number.

[8] A node that has no children

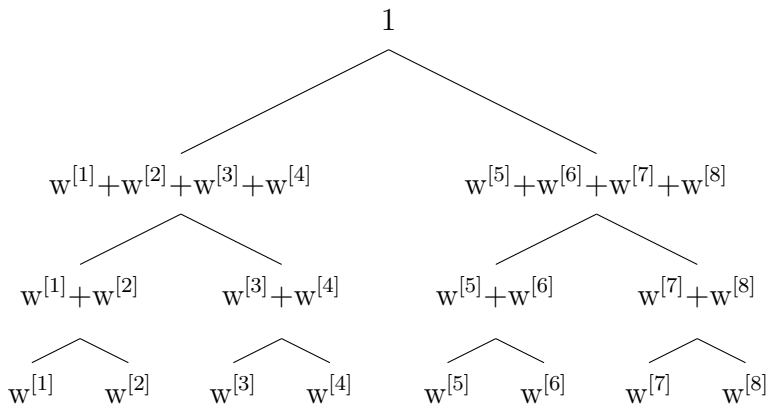
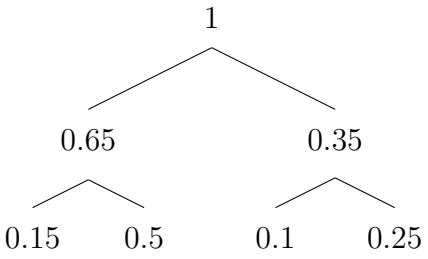


Figure 3.2: Binary tree for 8 particles

When the tree is built, the particles for the new generation get selected by M uniform distributed random numbers, each of them used to choose one particle: depending on which leaf of the tree the random number falls in, the belonging particle is selected. This is best shown on an example using a set of 4 particles with weights $\mathbf{w} = [0.15, 0.5, 0.1, 0.25]^{\mathrm{T}}$, for which the generated tree is displayed in Figure 3.3a. The random numbers for the search and the selected particles are given in Table 3.1. In Figure 3.3b can be seen how the same tree looks in computer memory.



(a)

1	0.65	0.35	0.15	0.5	0.1	0.25

(b)

Figure 3.3: Example of a tree with 4 particles

Table 3.1: Example of resampling 4 particles

Random number	0.325	0.901	0.546	0.118
Selected particle	2	4	2	1

Algorithm 3.5 Resampling

Require: Set of M particles: P

Set of N_z measurements: z

Ensure: $N_{\text{eff}} < M/2$

$tree \leftarrow \textsc{BuildTree}(w_{\text{N}})$ ▷ Build tree from the weights of the particles

for $i \leftarrow 1, M$ **do**

$rand \leftarrow \textsc{RandUniform}(0, 1)$ ▷ Random number between 0 and 1

$j \leftarrow \textsc{SearchTree}(tree, rand)$ ▷ Index of the particle that survives

$newP^i \leftarrow P^j$ ▷ Copy the chosen particle to the new set

end for

$P \leftarrow newP$ ▷ The old set is replaced by the new one

3.6 Parallelization

In the algorithm a few things are parallelized for increasing it's performance. The changes do not introduce any changes in the way the algorithm works, the differences are in how certain things are computed and where. There are three important things that are executed in parallel:

1. All calculations for each particle
2. Sums of data from all particles (eg. the sum of weights, for normalization)
3. Binary tree generation and search

The other difference is where the algorithm is executed, and that is on the GPU of the PC. In section 4 it will be explained how it is possible to do so.

3.6.1 Parallelization at particle level

Parallelization at the particle level means that theoretically all particles are executed in parallel. Everywhere in the algorithm that a `for` loop that iterates through every particle is present, it is eliminated by parallel execution. In practice the degree of parallelization depends on the capabilities of the GPU and on the number of particles. For example, if the GPU is capable of running 512 parallel threads, and the number of particles is 1024, every thread performs sequentially the calculations for $1024/512 = 2$ particles: thread i, where $i \in \{1, \ldots, 512\}$, calculates for particles $P^{[i]}$ and $P^{[i+512]}$.

3.6.2 Sums

When there is the need to sum certain values from all the particles, instead of summing them sequentially, with a `for` loop, they are summed using the so called logarithmic reduction. It's called like that because of it's logarithmic complexity $\mathcal{O}(\log_2 n)$, where n is the number of elements (in this case M). The principle is shown in Figure 3.4 on a vector with 8 elements: each of the 3 steps is calculated in parallel.

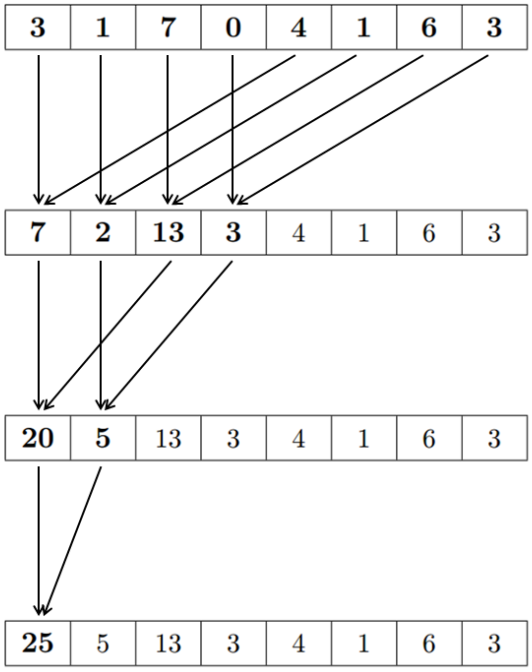


Figure 3.4: Logarithmic reduction

3.6.3 Building the tree

The binary tree is also created in parallel, in a very similar way to the logarithmic reduction. The principle is shown in Figure 3.5 on the tree from Figure 3.3. Each step is calculated in parallel.

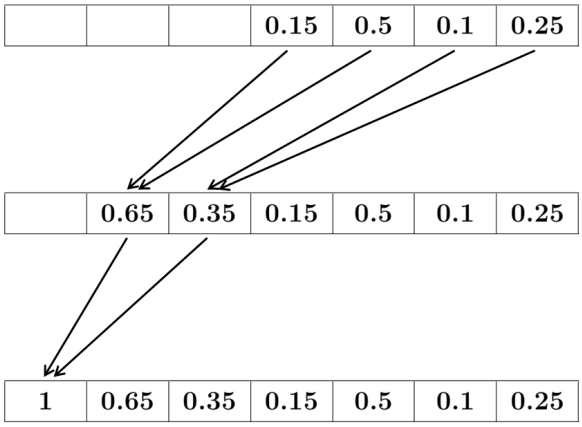


Figure 3.5: Building the binary tree

4 Nvidia CUDA

Nvidia CUDA C programming language has been chosen for the implementation of this algorithm. Compute Unified Device Architecture (CUDA) is Nvidia's architecture for GPUs that can perform both graphics rendering tasks and general purpose tasks. The language provided to program such GPU's is CUDA C, which is basically C with some specific additions.

Since the robot is normally programmed in NI LabVIEW, the code written in CUDA C is integrated as a dynamic-link library (.dll). A thing to consider in the future is that starting with LabVIEW 2012 the GPU Analysis Toolkit is available, which provides an interface to the CUDA Toolkit and the main libraries.

4.1 History of GPU computing

The application of GPUs in general purpose computing begins in 1999 with the launch of the Nvidia GeForce 256, less than a year before Nvidia coined the term Graphics Processing Unit (GPU). Not only game developers and artists were interested in this technology, but also a small number of researchers, who began to discover it's computing potentials. Like the subsequently released Nvidia GeForce 3 and ATI Radeon 8500, that GPU was of limited programmability. General purpose programming was very difficult, since graphics API like DirectX and OpenGL had to be used. Even for those familiar with those languages it wasn't easy, because a scientific calculation had to be mapped onto problems that could be represented by triangles and polygons. Despite all the difficulties, GPGPU (General Purpose GPU) computing was born.

The first big wave of adoption of this technology happened between 2003 and 2006, when the number of users was beginning to significantly increase, which led to the publication of a significant number of scientific papers on the matter. In that period the GPUs evolved further, improving eg. floating point operations performance, but the big disvantage of forcing users to code using graphics API remained present. The first solution to this problem was offered by Nvidia, in the form of the CUDA architecture. Their creation was a GPU with ALUs (Arithmetic-Logic Units) designed to support standard floating point operations and to accept general purpose oriented instructions, along with those specific for graphics. Another significant improvement was made in memory access and management. To offer an interface to all the new functionality, Nvidia engineers created the first programming language oriented to general purpose computing, based on the widespread C language: CUDA C.

The creation of such architecture opened the way into GPGPU computing to a huge number of users, for a very large number of different applications. This started the "boom" that is still going on today. CUDA was followed by AMD Stream, that became OpenCL, which today is a very valid alternative. At the moment CUDA is the best option available if you have an Nvidia card, because it already has a huge amount of libraries available, but in the future there will be more and more different options, so it will be interesting to continue following the argument.

4.2 GPU architecture

Typical general-purpose graphics processors consist of multiple stream processors or CUDA cores grouped in units called Stream Multiprocessors (SM). Every CUDA core in a SM executes a thread. Groups of threads called blocks represent the number of threads running on a single SM. Every thread has access to Global memory, that is the largest but slowest one, and is used to transfer data to and from the GPU. All the threads running on the same SM can access a memory called Shared memory, that is physically close to the SM and it's much faster than the Global memory, but it can't be accessed from threads outside a block (running on different SM-s). When a variable is declared in this memory, identical instances of the variable are created in each SM's Shared memory and have no connection between them. Every variable declared in a kernel (GPU part of code) without special keywords is created in registers, which are individual for each stream processor (thread) and are also very fast to access. Groups of 32 consecutive threads are called warps. Threads within a warp are always synchronized, they act as a SIMD (Single Instruction Multiple Data) unit, while each warp in a block performs as a Simultaneous Multithreading (SMT) unit. A special function, `__syncthreads()`, is provided to create a synchronization point for all the threads, or more precisely for all the warps, within a block.

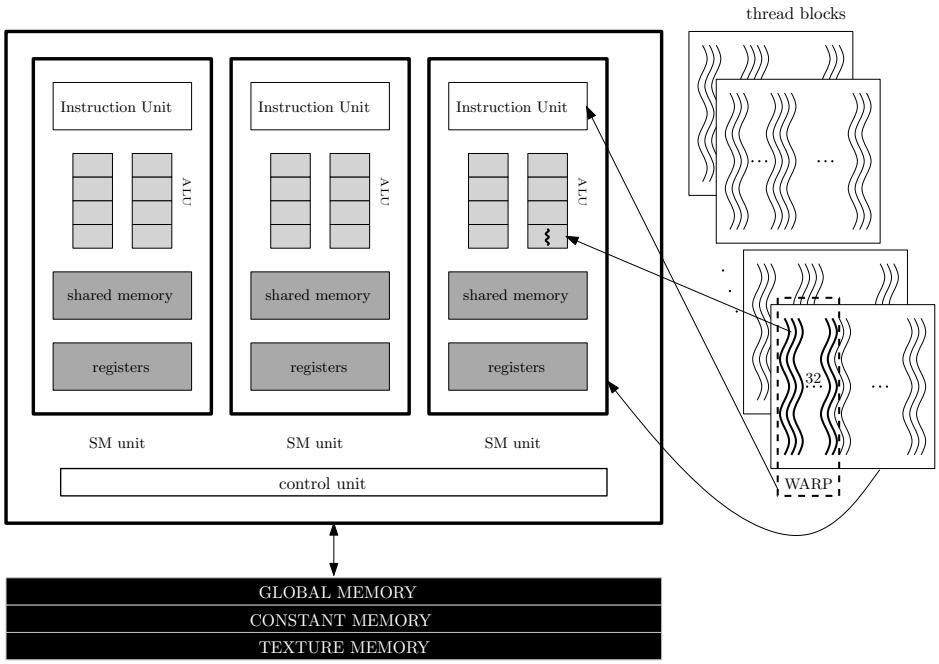


Figure 4.1: CUDA architecture [1]

Architectures like the one described are typically called SIMT (Single Instruction Multiple Threads). SIMT is considered a relax SIMD since threads can execute different instructions. However, different instructions within a warp can't execute in parallel, so they are serialized, which causes performance losses. If, for example, there is an `if-else` statement, the threads that must run the `else` part have to wait until the rest of the threads complete the `if`.

4.3 Programming

Today, along with the mentioned C language extension, there is an increasing number of available programming languages capable of using CUDA, like C++, Fortran, .NET languages, Python, Java, Ruby etc. Since in this case the code had to be implemented as a dynamic-link library, there was no need for a higher level language, so plain CUDA C has been used.

One of the most specific ways in which CUDA C differs from C is a heterogeneous view of processor and memory resources. One hand, the CPU, called **host**, a serial processor that is used in the traditional C way. On the other hand, the GPU, referred to as **device**, that executes parallel **kernels** and has access to separate memory spaces. It's good to point out that the execution is asynchronous, which means that when the host calls some kernel, it doesn't wait for the device to finish before continuing with execution.

All needed for the complete understanding of the code in Appendix A are a few additions to standard C made for CUDA.

4.3.1 Kernels

A kernel is defined as a normal function, with the addition of the keyword `__global__` in front of it. In the kernel call, performed from the host, two arguments are passed in three pairs of angle brackets:

```
demoKernel<<<BLOCKS, THREADS>>>();
```

This pair of arguments define how the kernel is executed, i.e. on how many blocks with how many threads each. The grid of blocks can be two-dimensional, while blocks can be three-dimensional. To be able to use all the dimensions, `THREADS` and `BLOCKS` are of the built-in `dim3` type. It creates a

simple structure containing `x`, `y` and `z`. One important difference kernels has from typical functions is that they must return `void`.

4.3.2 Memory allocation

If the kernel has to perform any "useful" operation, some memory has to be allocated on the device. For this purpose the function `cudaMalloc()` is used, it works just like `malloc()`, with the difference that it allocates in the global memory of the device.

To free the allocated memory, `cudaFree()` is used, again in the same way like `free()`.

To copy data between host and device there is `cudaMemcpy()`. It's like `memcpy()` but with an additional argument that defines the direction of data flow. This can be `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost` and `cudaMemcpyDeviceToDevice`.

4.3.3 Shared memory and thread synchronization

Like it was mentioned, shared memory is located physically close to the SM, it's very fast and it's not accessible outside of a block. To declare a variable or array in shared memory, the keyword `__shared__` must be in front of the declaration. A copy of the declared variable is created for every launched block. If the shared memory array size has to be defined at runtime, a third argument must be provided in the angle brackets: the desired memory size, and before the array declaration there must be the `extern` keyword. An important observation is that all pointers to dynamically allocated shared memory are given the same address, so the solution if you want to have two

different arrays is to allocate the overall amount of memory, then point the first pointer at the beginning of shared (address 0) and the second to the beginning increased by the first array's length.

For thread synchronization within a block, `__syncthreads()` is used. The best method to synchronize between different blocks, if can't be avoided, is to exit the kernel and launch a new one.

4.3.4 Device functions

Unlike `__global__` functions, `__device__` functions can only be called from device code. Their purpose is like that of standard functions – to separate and re-use some part of code. They are also called like standard functions, without any addition, and unlike kernels, can return any type of data. During runtime, every thread gets it's own copy of the function, so it's the equivalent of having code written in the caller part, the same behaviour of host functions.

4.3.5 Constants

CUDA offers some built-in constants that are useful for various purposes, like determining which thread has to do what. Some of those constants are:

- `gridDim` → structure containing the grid dimensions
- `blockIdx` → block index within the grid
- `blockDim` → block dimensions
- `threadIdx` → thread index within the block
- `warpSize` → warp size in number of threads

5 Results and final words

The algorithm has been tested in a laboratory, using reflective poles as static landmarks. During exploration the robot was able to successfully detect all the landmarks, discard false measurements and ignore moving obstacles (people) that could cover some landmark. When operating in the explored environment, the robot maintains a localization accuracy of 10 cm. The serial and the parallel implementations have been confronted, and the results are shown in Figure 1.1. The test was conducted using 1000 particles and with from 0 to 8 landmarks. From this comparison it can be seen how much faster is the execution on the GPU when bigger calculations are involved.

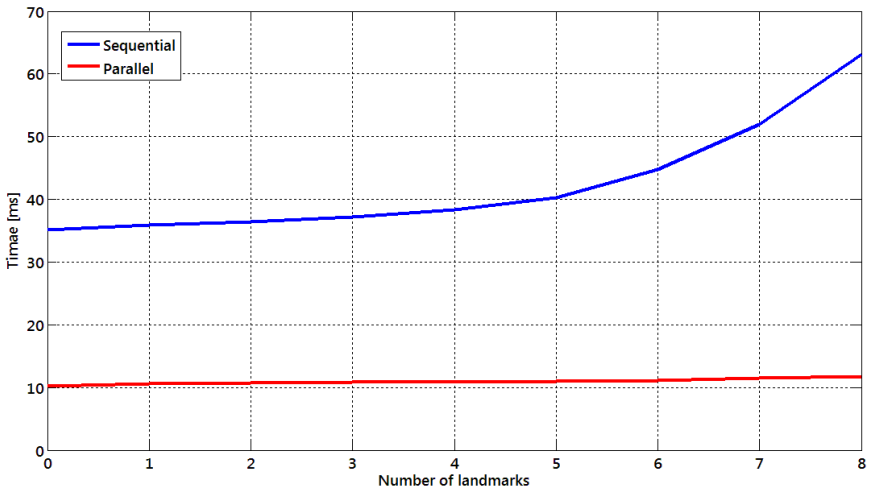


Figure 5.1: Comparison betwen sequential and parallel algortims

Testing and measurements are still in process, as the plan is to continue developing algorithms based on this concept, so collecting as much data as possible will help to understand better the weaknesses. In future it is planned to cover also the navigation segment, chasing the initially mentioned goal of

full autonomy of the robot.

This algorithm is a simple representation of the ideas behind particle filtering based simultaneous localization and mapping, and is not meant compete with the latest solutions. However, it served very well as an introduction to this field of robotics, I have acquired a large amount of knowledge during and after it's development, and it is my wish to continue working in this area.

References

[1] S. Jelić, S. Laue, D. Matijević, and P. Wijerama, "Fast parallel implementation of fractional packing and covering linear programs," Tech. Rep. MSU-CSE-00-2, Department of Mathematics, University of J.J. Strossmayer in Osijek, Trg Lj. Gaja 6, Osijek, September 2012.

[2] R. Smith, M. Self, and P. Cheeseman, "Estimating uncertain spatial relationships in robotics," in *Proceedings of the Second Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI-86)*, (Corvallis, Oregon), pp. 267–288, AUAI Press, 1986.

[3] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit, "FastSLAM: A factored solution to the simultaneous localization and mapping problem," in *Proceedings of the AAAI National Conference on Artificial Intelligence*, (Edmonton, Canada), AAAI, 2002.

[4] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit, "FastSLAM 2.0: An improved particle filtering algorithm for simultaneous localization and mapping that provably converges," in *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI)*, (Acapulco, Mexico), IJCAI, 2003.

[5] I. M. Rekleitis, "A particle filter tutorial for mobile robot localization," Tech. Rep. TR-CIM-04-02, Centre for Intelligent Machines, McGill University, 3480 University St., Montreal, Québec, CANADA H3A 2A7, 2004.

[6] S. Thrun, "Particle filters in robotics," in *Proceedings of the 17th Annual Conference on Uncertainty in AI (UAI)*, 2002.

[7] N. J. Gordon, D. J. Salmond, and A. F. M. Smith, "Novel approach to nonlinear/non-Gaussian Bayesian state estimation," *Radar and Signal Processing, IEE Proceedings F*, vol. 140, pp. 107–113, Apr. 1993.

[8] S. M. Bozic, *Digital and Kalman Filtering*. London: E. Arnold, 1979.

[9] S. I. Roumeliotis and G. A. Bekey, "Bayesian estimation and Kalman filtering: A unified framework for mobile robot localization," in *Robotics and Automation, 2000. Proceedings. ICRA '00. IEEE International Conference on*, vol. 3, pp. 2985–2992 vol.3, IEEE, 2000.

[10] R. C. Smith and P. Cheeseman, "On the representation and estimation of spatial uncertainly," *Int. J. Rob. Res.*, vol. 5, pp. 56–68, Dec. 1986.

[11] A. Doucet, N. de Freitas, K. Murphy, and S. Russell, "Rao-Blackwellised Particle Filtering for Dynamic Bayesian Networks," in *Conference on Uncertainty in Artificial Intelligence*, pp. 176–183, 2000.

[12] K. Murphy, "Bayesian map learning in dynamic environments," in *Advances In Neural Information Processing Systems 12*, MIT Press, 1999.

A Source code

This code is written to be able to run on devices with the CUDA compute capability 1.2. Using devices with newer architecture some of the described steps could be done more elegantly and easier, eg. using dynamic device memory allocation. The code is written to be a dynamic link library (dll), and the main loop of the whole algorithm is outside it (in this case the dll is called from LabVIEW).

```
1  /*
2  -
3  -   Particle filter SLAM with CUDA
4  -   Author: Matija Rossi
5  -
6  */
7
8  #include <math.h>
9  #include <time.h>
10 #include <cuda.h>
11 #include <cuda_runtime.h>
12 #include <device_launch_parameters.h>
13 #include <curand.h>
14 #include <curand_kernel.h>
15 #include <iostream>
16
17 #define SIGMA_ANG   0.08f
18 #define SIGMA_RAD   1.0f
19 #define SIGMA_LW    0.05f
20 #define SIGMA_RW    0.05f
21 #define SM_ANG      0.005f
22 #define SM_R        0.5f
23
```

```
24 using namespace std;
25
26
27 typedef struct measurement {    // laser measurements
28     float thetaM;
29     float distM;
30 } measurement;
31
32 typedef struct landmark {   // landmarks for every particle
33     int counterL;
34     float xL;
35     float yL;
36     int lastSeenL;
37 } landmark;
38
39 typedef struct particle {   // particles
40     float thetaP;
41     float xP;
42     float yP;
43     float laserxP;
44     float laseryP;
45     landmark landP[20];
46     int landNumberP;
47     float probP;
48 } particle;
49
50 typedef struct center { // weighted mean of particles
51     float thetaC;
52     float xC;
53     float yC;
54 } center;
55
56
```

```
57 int powerOf2(int n);
58 extern "C" void __declspec(dllexport) __cdecl SLAM(float
      *pCent, float fiR, float fiL, float period, unsigned long
      iteration, int nM, float *me, float *l);
59 extern "C" void __declspec(dllexport) __cdecl
      withoutMeasurements(float *pCent, float fiR, float fiL,
      float period);
60 extern "C" int __declspec(dllexport) __cdecl init(int n);
61 extern "C" void __declspec(dllexport) __cdecl freeP(int
      input);
62
63 __device__ void particleLaserDistance(int tid, particle *p);
64 __device__ void dataAssociation(int tid, particle *p,
      measurement *m, int nM, unsigned long iteration);
65 __device__ float angleAdjust(float angle);
66 __device__ void sort(measurement *m, int n);
67 __device__ void warpSum(volatile float *sum, int tid);
68 __device__ void warpSumC(volatile center *cent, int tid);
69 __global__ void initializeParticles(particle *p1, particle
      *p2, int nP, curandState * state, unsigned long seed);
70 __global__ void slam1(particle *p, measurement *m, int nP,
      int nM, unsigned long iteration, float *s, float *fi,
      float period, curandState *globalState);
71 __global__ void weightsVariance(particle *p, int nP, float
      *s, int b1);
72 __global__ void slam2(particle *p, particle *pNew, center
      *pCenter, int nP, float *s, int b1, float *binTree, int
      nP2);
73 __global__ void resampling(particle *p, particle *pNew,
      float *binTree, int nP, int nP2, curandState
      *globalState);
74 __global__ void noMeasurements(particle *p, int nP, float
      period, float *fi, curandState *globalState, center
```

```
    *pCenter);
75
76
77  /* ------------------------------------------------------ */
78  /* --------------------- DEVICE CODE -------------------- */
79  /* ------------------------------------------------------ */
80
81
82  __global__ void initializeParticles(particle *p1, particle
        *p2, int nP, curandState * state, unsigned long seed) {
83      int tid = blockIdx.x*blockDim.x + threadIdx.x;
84      if(tid < nP) {
85          curand_init(seed, tid, 0, &state[tid]);
86          p1[tid].thetaP = 1.5707963f;
87          p1[tid].xP = 0;
88          p1[tid].yP = 0;
89          p1[tid].landNumberP = 0;
90          p1[tid].probP = 1.0f/nP;
91          p2[tid].thetaP = 1.5707963f;
92          p2[tid].xP = 0;
93          p2[tid].yP = 0;
94          p2[tid].landNumberP = 0;
95          p2[tid].probP = 1.0f/nP;
96      }
97  }
98
99  __device__ void particleLaserDistance(int tid, particle *p) {
100     p[tid].laserxP = p[tid].xP + (49*cosf(p[tid].thetaP));
101     p[tid].laseryP = p[tid].yP + (49*sinf(p[tid].thetaP));
102 }
103
104 __device__ void dataAssociation(int tid, particle *p,
        measurement *m, int nM, unsigned long iteration) {
```

```
105     for(int i=0; i<nM; i++) {
106         //float angN =
                curand_normal(&globalState[tid])*SM_ANG, radN =
                curand_normal(&globalState[tid])*SM_R;
107         float measuredCoord[2];
108         measuredCoord[0] = p[tid].laserxP +
               m[i].distM*cosf(m[i].thetaM + p[tid].thetaP);
109         measuredCoord[1] = p[tid].laseryP +
               m[i].distM*sinf(m[i].thetaM + p[tid].thetaP);
110         bool isAssociated = false;
111         for(int j=0; j<20; j++) {
112             if(p[tid].landP[j].counterL != 0 &&
                   sqrtf(powf(p[tid].landP[j].xL -
                   measuredCoord[0], 2) +
                   powf(p[tid].landP[j].yL - measuredCoord[1],
                   2)) <= 100) {
113                 p[tid].landP[j].xL = (p[tid].landP[j].xL *
                       p[tid].landP[j].counterL +
                       measuredCoord[0]) /
                       (p[tid].landP[j].counterL + 1);
114                 p[tid].landP[j].yL = (p[tid].landP[j].yL *
                       p[tid].landP[j].counterL +
                       measuredCoord[1]) /
                       (p[tid].landP[j].counterL + 1);
115                 p[tid].landP[j].counterL++;
116                 p[tid].landP[j].lastSeenL = iteration;
117                 isAssociated = true;
118                 break;
119             }
120         }
121         if(!isAssociated) {
122             int i = 0;
123             bool empty = false;
```

```
124             for(i=0; i<20; i++)
125                 if(p[tid].landP[i].counterL == 0) {
126                     empty = true;
127                     break;
128                 }
129             if(empty) {
130                 p[tid].landP[i].xL = measuredCoord[0];
131                 p[tid].landP[i].yL = measuredCoord[1];
132                 p[tid].landP[i].counterL = 1;
133                 p[tid].landP[i].lastSeenL = iteration;
134                 p[tid].landNumberP++;
135             }
136         }
137         for(int j=0; j<20; j++) {
138             if(p[tid].landP[j].counterL != 0 &&
                   (p[tid].landP[j].counterL < 100 && (iteration
                   - p[tid].landP[j].lastSeenL) > 50)) {
139                 p[tid].landP[j].counterL = 0;
140                 p[tid].landNumberP--;
141             }
142         }
143     }
144 }
145
146 __device__ float angleAdjust(float angle) {
147     while(angle > 3.141593f) angle -= 6.283185f;
148     while(angle < -3.141539f) angle += 6.283185f;
149     return angle;
150 }
151
152 __device__ void sort(measurement *m, int n) {
153     measurement swap;
154     int gap = n;
```

```
155     bool swapped = false;
156     while ((gap > 1) || swapped) {
157         if (gap > 1) gap = (int)(gap/1.247331f);
158         swapped = false;
159         for (int i=0; gap + i < n; ++i) {
160             if (m[i].thetaM - m[i + gap].thetaM > 0) {
161                 swap = m[i];
162                 m[i] = m[i + gap];
163                 m[i + gap] = swap;
164                 swapped = true;
165             }
166         }
167     }
168 }
169 
170 __device__ void warpSum(volatile float *sum, int tid) {
171     sum[tid] += sum[tid + 32];
172     sum[tid] += sum[tid + 16];
173     sum[tid] += sum[tid + 8];
174     sum[tid] += sum[tid + 4];
175     sum[tid] += sum[tid + 2];
176     sum[tid] += sum[tid + 1];
177 }
178 
179 __device__ void warpSumC(volatile center *cent, int tid) {
180     cent[tid].thetaC += cent[tid + 32].thetaC;
181     cent[tid].xC += cent[tid + 32].xC;
182     cent[tid].yC += cent[tid + 32].yC;
183 
184     cent[tid].thetaC += cent[tid + 16].thetaC;
185     cent[tid].xC += cent[tid + 16].xC;
186     cent[tid].yC += cent[tid + 16].yC;
187 
```

```
188     cent[tid].thetaC += cent[tid + 8].thetaC;
189     cent[tid].xC += cent[tid + 8].xC;
190     cent[tid].yC += cent[tid + 8].yC;
191
192     cent[tid].thetaC += cent[tid + 4].thetaC;
193     cent[tid].xC += cent[tid + 4].xC;
194     cent[tid].yC += cent[tid + 4].yC;
195
196     cent[tid].thetaC += cent[tid + 2].thetaC;
197     cent[tid].xC += cent[tid + 2].xC;
198     cent[tid].yC += cent[tid + 2].yC;
199
200     cent[tid].thetaC += cent[tid + 1].thetaC;
201     cent[tid].xC += cent[tid + 1].xC;
202     cent[tid].yC += cent[tid + 1].yC;
203 }
204
205 __global__ void slam1(particle *p, measurement *m, int nP,
        int nM, unsigned long iteration, float *s, float *fi,
        float period, curandState *globalState) {
206     int tid = blockIdx.x*blockDim.x + threadIdx.x;
207     extern __shared__ float sum[];
208     sum[threadIdx.x] = 0;
209
210     if(tid < nP) {
211         float weight = p[tid].probP;
212
213         // simulate particle movement
214         float ln =
                curand_normal(&globalState[tid])*SIGMA_LW, rn =
                curand_normal(&globalState[tid])*SIGMA_RW;
215         float sp = 5*(fi[0] + fi[1] + fi[0]*rn + fi[1]*ln);
216         float ang = 0.0943396f*(fi[0] - fi[1] + fi[0]*rn -
```

```
            fi[1]*ln);
217         p[tid].xP += cosf(p[tid].thetaP + ang)*sp;
218         p[tid].yP += sinf(p[tid].thetaP + ang)*sp;
219         p[tid].thetaP = angleAdjust(p[tid].thetaP + 2*ang);
220
221         particleLaserDistance(tid, p);
222
223         // update map
224         dataAssociation(tid, p, m, nM, iteration);
225
226         // simulate particle measurements
227         measurement measP[20];
228         int nMeasP = 0;
229         float angN =
               curand_normal(&globalState[tid])*SM_ANG, radN =
               curand_normal(&globalState[tid])*SM_R;
230         for(int i=0; i<20; i++) {
231             if(p[tid].landP[i].counterL != 0) {
232                 measP[nMeasP].distM =
                       sqrtf(powf(p[tid].landP[i].xL -
                       p[tid].laserxP, 2) +
                       powf(p[tid].landP[i].yL - p[tid].laseryP,
                       2)) + radN;
233                 measP[nMeasP].thetaM =
                       angleAdjust(atan2f(p[tid].landP[i].yL -
                       p[tid].laseryP, p[tid].landP[i].xL -
                       p[tid].laserxP) - p[tid].thetaP + angN);
234                 if(measP[nMeasP].distM <= 350 &&
                       abs(measP[nMeasP].thetaM) <= 1.5708f)
                       nMeasP++;
235             }
236         }
237         sort(measP, nMeasP);
```

```
238
239         // correct real measurements array
240         measurement measR[20];
241         int count = 0;
242         for(int i=0; i<nMeasP; i++) {
243             if(abs(measP[i].thetaM - m[i].thetaM) > 0.3f ||
                    abs(measP[i].distM - m[i].distM) > 30) {
244                 measR[i].thetaM = -10;
245                 measR[i].distM = -10;
246                 count++;
247                 continue;
248             }
249             measR[i] = m[i-count];
250         }
251
252         // calculate particle weights
253         for(int i=0; i<nMeasP; i++) {
254             if(measR[i].thetaM != -10 && measR[i].distM !=
                    -10) {
255                 float ni = cosf(measR[i].thetaM) *
                            cosf(measP[i].thetaM) +
                            sinf(measR[i].thetaM) *
                            sinf(measP[i].thetaM);
256                 if(ni>1) ni = 1;
257                 if(ni<-1) ni=-1;
258                 weight *=
                            expf(-0.5f*powf(acosf(ni)/SIGMA_ANG,
                            2))/(SIGMA_ANG*2.5066283f)*
259                             expf(-0.5f*powf((measP[i].distM -
                                measR[i].distM)/(SIGMA_RAD*100.0f),
                                2))/(SIGMA_RAD*2.5066283f);
260             }
261         }
```

```
262         if(weight < 0.00000001f) weight = 0.00000001f;
263         sum[threadIdx.x] = weight;
264         p[tid].probP = weight;
265         __syncthreads();
266         int iSum = blockDim.x/2;
267         while (iSum > 32) {
268             if (threadIdx.x < iSum) sum[threadIdx.x] +=
                    sum[threadIdx.x + iSum];
269             __syncthreads();
270             iSum /= 2;
271         }
272         volatile float *s_p = sum;
273         warpSum(s_p, threadIdx.x);
274         if(threadIdx.x == 0) s[blockIdx.x] = sum[0];
275     }
276
277     // end of kernel for block sinchronization
278 }
279
280 __global__ void weightsVariance(particle *p, int nP, float
        *s, int b1) {
281     int tid = blockIdx.x*blockDim.x + threadIdx.x;
282     extern __shared__ float vars[];
283     vars[threadIdx.x] = 0;
284
285     // calculate overall sum of particle weights
286     if(tid < nP) {
287         float sum = 0;
288         for(int i=0; i<b1; i++)
289             sum += s[i];
290
291         // particle probability (normalized)
292         p[tid].probP /= sum;
```

```
293
294         vars[threadIdx.x] = powf(p[tid].probP, 2);
295         __syncthreads();
296         int iVar = blockDim.x/2;
297         while (iVar > 32) {
298             if (threadIdx.x < iVar) vars[threadIdx.x] +=
                    vars[threadIdx.x + iVar];
299             __syncthreads();
300             iVar /= 2;
301         }
302         volatile float *v_p = vars;
303         warpSum(v_p, threadIdx.x);
304         if(threadIdx.x == 0) s[blockIdx.x] = vars[0];
305     }
306 }
307
308 __global__ void slam2(particle *p, particle *pNew, center
    *pCenter, int nP, float *v, int b1, float *binTree, int
    nP2) {
309     int tid = blockIdx.x*blockDim.x + threadIdx.x;
310     extern __shared__ center cent[];
311     cent[threadIdx.x].thetaC = 0; cent[threadIdx.x].xC = 0;
        cent[threadIdx.x].yC = 0;
312     float var = 0;
313
314     if(tid < nP) {
315
316         for(int i=0; i<b1; i++)
317             var += v[i];
318         var = 1.0f/var;
319         if(tid == 0)
320             v[0] = var;
321
```

```
322         // calculate center of particles
323         cent[threadIdx.x].thetaC =
               p[tid].thetaP*p[tid].probP;
324         cent[threadIdx.x].xC = p[tid].xP*p[tid].probP;
325         cent[threadIdx.x].yC = p[tid].yP*p[tid].probP;
326         __syncthreads();
327         int iCent = blockDim.x/2;
328         while (iCent > 32) {
329             if (threadIdx.x < iCent) {
330                 cent[threadIdx.x].thetaC += cent[threadIdx.x
                       + iCent].thetaC;
331                 cent[threadIdx.x].xC += cent[threadIdx.x +
                       iCent].xC;
332                 cent[threadIdx.x].yC += cent[threadIdx.x +
                       iCent].yC;
333             }
334             __syncthreads();
335             iCent /= 2;
336         }
337         volatile center *c_p = cent;
338         warpSumC(c_p, threadIdx.x);
339         if(threadIdx.x == 0)
340             pCenter[blockIdx.x] = cent[0];
341     }
342
343     if(var <= nP/2.0f && blockIdx.x == 0) {
344
345         // build binary tree
346         int idTree = threadIdx.x;
347         while(idTree < nP) {
348             binTree[idTree + nP2 - 1] = p[idTree].probP;
349             idTree += blockDim.x;
350         }
```

```
351         while(idTree < nP2) {
352             binTree[idTree + nP2 - 1] = 0;
353             idTree += blockDim.x;
354         }
355         __syncthreads();
356         int n1, n2, iT = 0;
357         n1 = nP2/blockDim.x;
358         n1 = (n1==0 ? 0:(n1-1));
359         idTree = threadIdx.x - 1 + (n1*blockDim.x);
360         while(iT < n1) {
361             binTree[idTree] = binTree[idTree*2 + 1] +
                    binTree[(idTree*2) + 2];
362             idTree -= blockDim.x;
363             iT++;
364             __syncthreads();
365         }
366         n2 = (nP2<blockDim.x ? nP2:blockDim.x);
367         for(int j=n2/2; j>0; j/=2) {
368             if(idTree >= j-1 && idTree < (j*2)-1)
369                 binTree[idTree] = binTree[idTree*2 + 1] +
                        binTree[idTree*2 + 2];
370             __syncthreads();
371         }
372     }
373     else if(var > nP/2 && tid < nP) {
374         pNew[tid] = p[tid];
375     }
376
377     // end of kernel for block sinchronization
378 }
379
380 __global__ void resampling(particle *p, particle *pNew,
       float *binTree, int nP, int nP2, curandState
```

```
    *globalState) {
381     int tid = blockIdx.x*blockDim.x + threadIdx.x;
382 
383     if(binTree[0] >= 1) {
384 
385         // binary search - new generation of particles
386         if(tid%32 == 0 && tid/32 < nP) {
387             float random =
                    curand_uniform(&globalState[tid/32]);
388             int iTr = 2;
389             while(iTr < nP2) {
390                 if(random < binTree[iTr - 1]) iTr *= 2;
391                 else {
392                     random -= binTree[iTr - 1];
393                     iTr = (iTr + 1)*2;
394                 }
395             }
396             if(random >= binTree[iTr - 1]) iTr++;
397             pNew[tid/32] = p[iTr - nP2];
398             pNew[tid/32].probP = 1.0f/nP;
399         }
400     }
401 }
402 
403 __global__ void noMeasurements(particle *p, int nP, float
    period, float *fi, curandState *globalState, center
    *pCenter) {
404     int tid = blockIdx.x*blockDim.x + threadIdx.x;
405     extern __shared__ center cent[];
406     cent[threadIdx.x].thetaC = 0; cent[threadIdx.x].xC = 0;
        cent[threadIdx.x].yC = 0;
407 
408     if(tid < nP) {
```

```
409
410         // set equal probabilities
411         float prob = p[tid].probP;
412
413         // simulate particle movement
414         float ln =
               curand_normal(&globalState[tid])*SIGMA_LW, rn =
               curand_normal(&globalState[tid])*SIGMA_RW;
415         float sp = 5*(fi[0] + fi[1] + fi[0]*rn + fi[1]*ln);
416         float ang = 0.0943396f*(fi[0] - fi[1] + fi[0]*rn -
               fi[1]*ln);
417         p[tid].xP += cosf(p[tid].thetaP + ang)*sp;
418         p[tid].yP += sinf(p[tid].thetaP + ang)*sp;
419         p[tid].thetaP = angleAdjust(p[tid].thetaP + 2*ang);
420
421         // calculate center of particles
422         cent[threadIdx.x].thetaC = p[tid].thetaP*prob;
423         cent[threadIdx.x].xC = p[tid].xP*prob;
424         cent[threadIdx.x].yC = p[tid].yP*prob;
425         __syncthreads();
426         int iCent = blockDim.x/2;
427         while (iCent > 32) {
428             if (threadIdx.x < iCent) {
429                 cent[threadIdx.x].thetaC += cent[threadIdx.x
                       + iCent].thetaC;
430                 cent[threadIdx.x].xC += cent[threadIdx.x +
                       iCent].xC;
431                 cent[threadIdx.x].yC += cent[threadIdx.x +
                       iCent].yC;
432             }
433             __syncthreads();
434             iCent /= 2;
435         }
```

```
436         volatile center *c_p = cent;
437         warpSumC(c_p, threadIdx.x);
438         if(threadIdx.x == 0) pCenter[blockIdx.x] = cent[0];
439     }
440 }
441
442
443
444 /* ------------------------------------------------------ */
445 /* ---------------------- HOST CODE --------------------- */
446 /* ------------------------------------------------------ */
447
448
449 curandState *devStates;
450 particle *p1_D, *p2_D, *p_D, *pNew_D;
451 int nP, nP2, nResample, t1, b1, t2, b2;
452 bool select;
453 center *pCenter, *pCenter_D;
454 float *sum_D, *tree_D, *fi_D, *weightsVar;
455 size_t sumSize, centerSize;
456
457
458 extern "C" void __declspec(dllexport) __cdecl SLAM(float
       *pCent, float fiR, float fiL, float period, unsigned long
       iteration, int nM, float *me, float *l) {
459     dim3 threads1(t1), blocks1(b1);
460     dim3 threads2(t2), blocks2(b2);
461     p_D = (select==true ? p1_D:p2_D);
462     pNew_D = (select==true ? p2_D:p1_D);
463
464     float fi[2] = {fiR, fiL};
465
466     cudaMemcpy(fi_D, fi, 2*sizeof(float),
```

```
        cudaMemcpyHostToDevice);
467
468     measurement *m, *m_D;
469     m = (measurement *)malloc(nM*sizeof(measurement));
470     for(int i=0; i<2*nM; i++) {
471         if(i%2 == 0) m[i/2].thetaM = me[i];
472         else m[(i - 1)/2].distM = me[i];
473     }
474
475     cudaMalloc((void **)&m_D, nM*sizeof(measurement));
476     cudaMemcpy(m_D, m, nM*sizeof(measurement),
        cudaMemcpyHostToDevice);
477
478     slam1<<<blocks1, threads1, sumSize>>>(p_D, m_D, nP, nM,
        iteration, sum_D, fi_D, period, devStates);
479     weightsVariance<<<blocks1, threads1, sumSize>>>(p_D, nP,
        sum_D, b1);
480     slam2<<<blocks1, threads1, centerSize>>>(p_D, pNew_D,
        pCenter_D, nP, sum_D, b1, tree_D, nP2);
481     cudaMemcpy(weightsVar, sum_D, sizeof(float),
        cudaMemcpyDeviceToHost);
482     if(weightsVar[0] < nResample)
483         resampling<<<blocks2, threads2>>>(p_D, pNew_D,
            tree_D, nP, nP2, devStates);
484
485     cudaMemcpy(pCenter, pCenter_D, b1*sizeof(center),
        cudaMemcpyDeviceToHost);
486
487     for(int i=1; i<b1; i++) {
488         pCenter[0].thetaC += pCenter[i].thetaC;
489         pCenter[0].xC += pCenter[i].xC;
490         pCenter[0].yC += pCenter[i].yC;
491     }
```

```
492     pCent[0] = pCenter[0].thetaC;
493     pCent[1] = pCenter[0].xC;
494     pCent[2] = pCenter[0].yC;
495     for(int i=1; i<b1; i++) {
496         pCenter[i].thetaC = 0;
497         pCenter[i].xC = 0;
498         pCenter[i].yC = 0;
499     }
500 
501     select = (select==true ? false:true);
502 
503     cudaFree(m_D);
504     free(m);
505 }
506 
507 extern "C" void __declspec(dllexport) __cdecl
        withoutMeasurements(float *pCent, float fiR, float fiL,
        float period) {
508     dim3 threads(t1), blocks(b1);
509 
510     p_D = (select==true ? p1_D:p2_D);
511 
512     float fi[2] = {fiR, fiL};
513     cudaMemcpy(fi_D, fi, 2*sizeof(float),
            cudaMemcpyHostToDevice);
514 
515     noMeasurements<<<blocks, threads, centerSize>>>(p_D, nP,
            period, fi_D, devStates, pCenter_D);
516 
517     cudaMemcpy(pCenter, pCenter_D, b1*sizeof(center),
            cudaMemcpyDeviceToHost);
518 
519     for(int i=1; i<b1; i++) {
```

```
520         pCenter[0].thetaC += pCenter[i].thetaC;
521         pCenter[0].xC += pCenter[i].xC;
522         pCenter[0].yC += pCenter[i].yC;
523     }
524 
525     pCent[0] = pCenter[0].thetaC;
526     pCent[1] = pCenter[0].xC;
527     pCent[2] = pCenter[0].yC;
528 
529     for(int i=1; i<b1; i++) {
530         pCenter[i].thetaC = 0;
531         pCenter[i].xC = 0;
532         pCenter[i].yC = 0;
533     }
534 }
535 
536 extern "C" int __declspec(dllexport) __cdecl init(int n)  {
537     dim3 blocks(n/512 + (n%512==0 ? 0:1)), threads(512);
538     nP = n;
539     nP2 = powerOf2(nP);
540     nResample = nP/2;
541     int x = nP2>64 ? nP2:64;
542     t1 = x<512 ? x:512; b1 = nP2/t1+(nP2%t1==0 ? 0:1);
543     t2 = 512; b2 = nP*32/512+(nP*32%512==0 ? 0:1);
544     sumSize = t1*sizeof(float);
545     centerSize = t1*sizeof(center);
546 
547     weightsVar = (float *)malloc(sizeof(float));
548     pCenter = (center *)malloc(b1*sizeof(center));
549 
550     cudaMalloc((void **)&p1_D, n*sizeof(particle));
551     cudaMalloc((void **)&p2_D, n*sizeof(particle));
552     cudaMalloc((void **)&devStates, n*sizeof(curandState));
```

```
553
554     cudaMalloc((void **)&pCenter_D, b1*sizeof(center));
555     cudaMalloc((void **)&fi_D, 2*sizeof(float));
556     cudaMalloc((void **)&sum_D, b1*sizeof(float));
557     cudaMalloc((void **)&tree_D, (2*nP2-1)*sizeof(float));
558
559     initializeParticles<<<blocks, threads>>>(p1_D, p2_D, nP,
            devStates, time(NULL));
560
561     select = true;
562
563     return 1;
564 }
565
566 extern "C" void __declspec(dllexport) __cdecl freeP(int
        input) {
567
568     cudaFree(p1_D); cudaFree(p2_D); cudaFree(devStates);
            cudaFree(fi_D); cudaFree(sum_D); cudaFree(tree_D),
            cudaFree(pCenter_D); cudaFree(p_D); cudaFree(pNew_D);
569     free(pCenter); free(weightsVar);
570 }
571
572 int powerOf2(int n) {
573     n--;
574     n |= n >> 1;
575     n |= n >> 2;
576     n |= n >> 4;
577     n |= n >> 8;
578     n |= n >> 16;
579     return n+1;
580 }
```